Tattoo Recognition Technology - Evaluation (Tatt-E)

A Public Evaluation of Tattoo Recognition Algorithms

Concept, Evaluation Plan, and API
Version 1.0

Mei Ngan and Patrick Grother

December 1, 2016

Status of this Document

This document is intended to be the final specification. Changes are marked in green. Comments and questions should be submitted to tatt-e@nist.gov.

Timeline of the Tatt-E Activity

API Development	September 26, 2016	Draft evaluation plan available for public comments
	December 1, 2016	Final evaluation plan published
Phase 1	December 1, 2016	Participation starts: Algorithms may be sent to NIST
	~~February~~ March 1, 2017	**Last day for submission of algorithms to Phase 1**
	~~March~~ April 14, 2017	Interim results released to Phase 1 participants
Phase 2	~~May~~ June 1, 2017	**Last day for submission of algorithms to Phase 2**
	~~June~~ July 14, 2017	Interim results released to Phase 2 participants
Phase 3	~~August~~ September 15, 2017	**Last day for submission of algorithms to Phase 3**
	Q4 2017	Release of final public report

Acknowledgements

The organizers would like to thank the sponsor of this activity, the Federal Bureau of Investigation (FBI) Biometric Center of Excellence (BCOE) for initiating and progressing this work.

Contact Information

Email: tatt-e@nist.gov

Tatt-E Website:

https://www.nist.gov/programs-projects/tattoo-recognition-technology-evaluation-tatt-e

Table of Contents

List of Tables

1. Tatt-E

1.1 Background

Tattoos have been used for many years to assist law enforcement in the identification of criminals and victims and for investigative research purposes. Historically, law enforcement agencies have followed the ANSI/NIST-ITL 1-2011[1] standard to collect and assign keyword labels to tattoos. This keyword labeling approach comes with drawbacks, which include the limited number of ANSI/NIST standard class labels to able describe the increasing variety of new tattoo designs, the need for multiple keywords to sufficiently describe some tattoos, and subjectivity in human annotation as the same tattoo can be labeled differently by examiners. As such, the shortcomings of keyword-based tattoo image retrieval have driven the need for automated image-based tattoo recognition capabilities.

1.2 The Tattoo Recognition Technology Program

The Tattoo Recognition Technology Program was initiated by NIST to support an operational need for image-based tattoo recognition to support law enforcement applications. The program provides quantitative support for tattoo recognition development and best practice guidelines. Program activities to date are summarized in **Figure 1**.

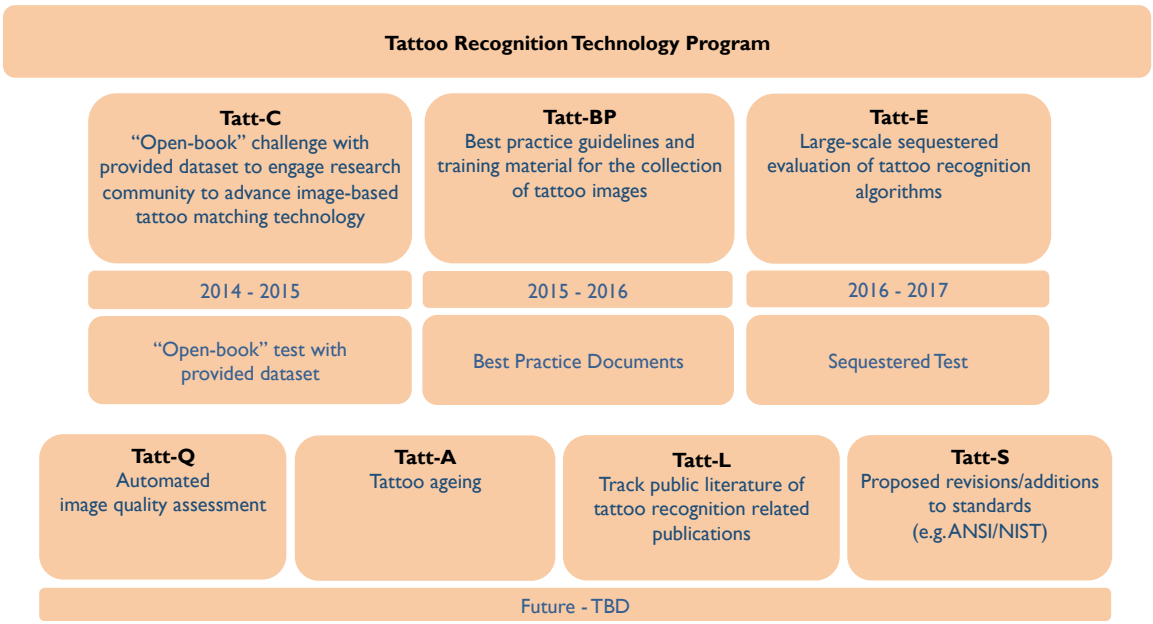

Figure 1 – Activities under the Tattoo Recognition Technology Program

- **Tatt-C** was an initial research challenge that provided operational data and use cases to the research community to advance research and development into automated image-based tattoo technologies and to assess the state-of-the-art. NIST hosted a culminating industry workshop and published a public report on the outcomes and recommendations from the Tatt-C activity. Please visit https://www.nist.gov/programs-projects/tattoo-recognition-technology-challenge-tatt-c for more information.

- **Tatt-BP** provides best practice guidance material for the proper collection of tattoo images to support image-based tattoo recognition. Recognition failure in Tatt-C was often related to the consistency and quality of image capture, and Tatt-BP aimed to provide guidelines on improving the quality of tattoo images collected operationally. Please visit https://www.nist.gov/itl/iad/image-group/tattoo-recognition-technology-best-practices for more information.

[1] The latest version of the ANSI/NIST-ITL 1-2011 standard is available at https://www.nist.gov/programs-projects/ansinist-itl-standard.

- **Tatt-E** is a sequestered evaluation intended to assess tattoo recognition algorithm accuracy and run-time performance over a large-scale of operational data. The participation details of Tatt-E are established in this document, also available for download at https://www.nist.gov/programs-projects/tattoo-recognition-technology-evaluation-tatt-e.

1.3 Scope

The Tattoo Recognition Technology – Evaluation (Tatt-E) is being conducted to assess and measure the capability of systems to perform automated image-based tattoo recognition. Both comparative and absolute accuracy measures are of interest, given the goals to determine which algorithms are most effective and viable for the following primary operational use-cases:

- Tattoo/Region of Interest Identification – matching different instances of the same tattoo image from the same subject over time. This includes matching with entire and/or partial regions of a tattoo.
- Tattoo detection/localization – determining whether an image contains a tattoo and if so, segmentation of the tattoo.
- Sketches – matching sketches to tattoo images.

Out of scope: Areas that are out of scope for this evaluation and will not be studied include: matching of tattoos based on thematically similar content as the definition of "similarity" is ill-defined; tattoo recognition in video.

This document establishes a concept of operations and an application programming interface (API) for evaluation of tattoo recognition implementations submitted to NIST's Tattoo Recognition Technology – Evaluation. See https://www.nist.gov/programs-projects/tattoo-recognition-technology-evaluation-tatt-e for all Tatt-E documentation.

1.4 Audience

Any person or organizations with capabilities in any of the following areas are invited to participation in the Tatt-E test.

- Tattoo matching implementations.
- Tattoo detection and localization algorithms.
- Algorithms with an ability to match sketches to tattoos.

Participants will need to implement the API defined in this document. Participation is open worldwide. There is no charge for participation. NIST encourages submission of experimental prototypes as well as those that could be readily made operational.

1.5 Training Data

None of the test data can be provided to participants. Instead prospective participants should leverage public domain and proprietary datasets as available. The Tatt-C dataset, which is provided by the FBI, is a very suitable tattoo corpus for development and training that has been made available to qualified developers - please contact tatt-e@nist.gov for more details.

1.6 Offline Testing

While Tatt-E is intended as much as possible to mimic operational reality, this remains an offline test executed on databases of images. The intent is to assess the core algorithmic capability of tattoo detection, localization, and recognition algorithms. Offline testing is attractive because it allows uniform, fair, repeatable, and efficient evaluation of the underlying technologies. Testing of implementations under a fixed API allows for a detailed set of performance related parameters to be measured.

1.7 Phased Testing

To support development, Tatt-E will run in multiple phases. The final phase will result in the release of public reports. Providers should not submit revised algorithms to NIST until NIST provides results for the prior phase.

For the schedule and number of algorithms of each class that may be submitted for each class, see section 1.12.

1.8 Interim reports

The performance of each implementation in phase 1 and 2 will be reported in a "report card". This will be provided to the participant. It is intended to facilitate research and development, not for marketing. Report cards will: be machine generated (i.e. scripted); be provided to participants with coded identification of their implementation; include timing, accuracy, and other performance results; include results from other implementations, but will not identify the other providers; be expanded and modified as revised implementations are tested and as analyses are implemented; be produced independently of the status of other providers' implementations; be regenerated on-the-fly, usually whenever any implementation completes testing, or when new analysis is added.

1.9 Final reports

NIST will publish one or more final public reports. NIST may also publish: additional supplementary reports (typically as numbered NIST Interagency Reports); in academic journal articles; in conferences and workshops (typically PowerPoint).

Our intention is that the final test reports will publish results for the best-performing implementation from each participant. Because "best" is underdefined (accuracy vs. time, for example), the published reports may include results for other implementations. The intention is to report results for the most capable implementations (see section 0 on metrics). Other results may be included (e.g. in appendices) to show, for example, illustration of progress or tradeoffs.

IMPORTANT: All Phase 3 results will be attributed to the providers, publicly associating performance with organization name.

1.10 Application scenarios

As described in Table 1, the test is intended to represent:

- Use of tattoo recognition technologies in search applications in which the enrolled dataset could contain images in the hundreds of thousands.
- Tattoo detection and localization with zero or more tattoos in the sample.

Table 1 – Subtests supported under the Tatt-E activity

#	Class label	D	I
1.	Aspect	Detection and Localization	1:N Search
2.	Enrollment dataset	None, application to single images	N enrolled subjects
3.	Prior NIST test references	For detection task, see Detection in Tatt-C 2015[2]	See Tattoo Identification, Region of Interest, and Mixed Media matching from Tatt-C 2015[2]
4.	Example application	Database construction and maintenance of large amounts of unlabeled, comingled data, e.g. given a pile of seized media, 1. Detect whether/which images contain tattoos and 2. Segment tattoos as pre-processing step for	Open-set search of a tattoo/sketch image against a central tattoo database, e.g. a search of a tattoo, parts of a tattoo, or a sketch of a tattoo against a tattoo database of known criminals.

[2] See the Tatt-C test report: NIST Interagency Report 8078, linked from https://www.nist.gov/programs-projects/tattoo-recognition-technology-challenge-tatt-c

		search against a database.	
5.	Number of images	Variable	Enrollment gallery: Up to $O(10^5)$
6.	Number of images per individual	N/A	Variable: one or more still tattoo images
7.	Enrollment image types	Tattoo and non-tattoo images	Tattoos
8.	Probe image types	N/A	Tattoos and sketches

1.11 Rules for participation

There is no charge to participate in Tatt-E. A participant must properly follow, complete, and submit the Participation Agreement ~~contained in this document~~ published on the Tatt-E website. This must be done once, after December 1, 2016. It is not necessary to do this for each submitted software library.

- All participants shall submit at least one class D (detection and localization) algorithm.
- Class I (identification) algorithms may be submitted only if at least 1 class D algorithm is also submitted.
- All submissions shall implement exactly one of the functionalities defined in Table 2. A library shall not implement the API of more than one class (separate libraries shall be submitted to participate in separate participation classes).

Table 2 – Tatt-E classes of participation

Function	Detection and Localization	Identification
Class label	D	I
Co-requisite class	None	D
API requirements	3.3	3.4

1.12 Number and schedule of submissions

The test is conducted in three phases, as scheduled on page 2. The maximum total (i.e. cumulative) number of submissions is regulated in Table 3. Participation in Phase 1 is not required for algorithm submission in Phase 2 and 3.

Table 3 – Cumulative total number of algorithms

#	Phase 1	Total over Phases 1 + 2	Total over Phases 1 + 2 + 3
All classes of participation	2	4	6 ~~if at least 1 was successfully executed by end of Phase 1~~ ~~2 otherwise~~

The numbers above may be increased as resources allow.

1.13 Core accuracy metrics

For identification testing, the test will target open-universe applications such as searching tattoo databases of known criminals (where the subject may or may not exist in the gallery) and closed-set tasks where subject is known to be in the database, e.g. in prison or corrections environments. Both score-based and rank-based metrics will be considered. Rank-based metrics are appropriate for one-to-many applications that employ human examiners to adjudicate candidate lists. Score based metrics are appropriate for cases where transaction volumes are too high for human adjudication or when false alarm rates must otherwise be low. Metrics include, false positive and negative identification rate (FPIR and FNIR) and cumulative match characteristic that can depend on threshold and rank.

For detection and localization, assessments of overlap between detected and examiner-determined tattoo area will be considered along with score-based metrics including false positive and negative detection rate.

1.14 Reporting template size

Because template size is influential on storage requirements and computational efficiency, this API supports measurement of template size. NIST will report statistics on the actual sizes of templates produced by tattoo recognition implementations submitted to Tatt-E. NIST may also report statistics on runtime memory and other compute-performance characteristics.

1.15 Reporting computational efficiency

As with other tests, NIST will compute and report accuracy. In addition, NIST will also report timing statistics for all core functions of the submitted API implementations. This includes feature extraction and 1:N matching. For an example of how efficiency might be reported, see the final report of the FRVT 2013 test[3].

1.16 Exploring the accuracy-speed trade-space

NIST will explore the accuracy vs. speed tradeoff for tattoo recognition algorithms running on a fixed platform. NIST will report both accuracy and speed of the implementations tested. While NIST cannot force submission of "fast vs. slow" variants, participants may choose to submit variants on some other axis (e.g. "experimental vs. mature") implementations.

1.17 Hardware specification

NIST intends to support highly optimized algorithms by specifying the runtime hardware. There are several types of computers that may be used in the testing. The following list gives some details about possible compute architectures:

- Dual Intel Xeon X5680 3.3 GHz CPUs (6 cores each)
- Dual Intel Xeon X7560 2.3 GHz CPUs (8 cores each)
- Dual Intel Xeon E5-2695 3.3 GHz CPUs (14 cores each; 56 logical CPUs total) with Dual NVIDIA Tesla K40 GPUs

Each CPU has 512K cache. The bus runs at 667 Mhz. The main memory is 192 GB Memory as 24 8GB modules. We anticipate that 16 processes can be run without time slicing, though NIST will handle all multiprocessing work via `fork()`. Participant-initiated multiprocessing is not permitted.

NIST is requiring use of 64-bit implementations throughout. This will support large memory allocation to support 1:N identification tasks. Note that while the API allows read access of the disk during the 1:N search, the disk is relatively slow, and I/O will be included in your run time.

All GPU-enabled machines will be running CUDA version 7.5. cuDNN v5 for CUDA 7.5 will also be installed on these machines. Implementations that use GPUs will only be run on GPU-enabled machines.

1.18 Operating system, compilation, and linking environment

The operating system that the submitted implementations shall run on will be released as a downloadable file accessible from http://nigos.nist.gov:8080/evaluations/CentOS-7-x86_64-Everything-1511.iso, which is the 64-bit version of CentOS 7.2 running Linux kernel 3.10.0.

For this test, Windows machines will not be used. Windows-compiled libraries are not permitted. All software must run under CentOS 7.2.

NIST will link the provided library file(s) to our C++ language test drivers. Participants are required to provide their library in a format that is dynamically-linkable using the C++11 compiler, g++ version 4.8.5.

A typical link line might be

```
g++ -std=c++11 -I. -Wall -m64 -o tatte tatte.cpp  -L.  –ltatte_Company_D_07
```

The Standard C++ library should be used for development. The prototypes from this document will be written to a file "tatte.h" which will be included via

```
#include <tatte.h>
```

[3] See the FRVT 2013 test report: NIST Interagency Report 8009, linked from http://face.nist.gov/frvt

The header files will be made available to implementers via https://github.com/usnistgov/tattoo.

All compilation and testing will be performed on x86_64 platforms. Thus, participants are strongly advised to verify library-level compatibility with g++ (on an equivalent platform) prior to submitting their software to NIST to avoid linkage problems later on (e.g. symbol name and calling convention mismatches, incorrect binary file formats, etc.).

Any and all dependencies on external dynamic/shared libraries not provided by CentOS 7.2 as part of the built-in "development" package must be provided as a part of the submission to NIST.

1.18.1 Library and Platform Requirements

Participants shall provide NIST with binary code only (i.e. no source code). The implementation should be submitted in the form of a dynamically-linked library file.

The core library shall be named according to Table 4. Additional dynamic libraries may be submitted that support this "core" library file (i.e. the "core" library file may have dependencies implemented in these other libraries).

Intel Integrated Performance Primitives (IPP) ® libraries are permitted if they are delivered as a part of the developer-supplied library package. It is the provider's responsibility to establish proper licensing of all libraries. The use of IPP libraries shall not prevent run on CPUs that do not support IPP. Please take note that some IPP functions are multithreaded and threaded implementations are prohibited.

NIST will report the size of the supplied libraries.

Table 4 – Implementation library filename convention

Form	libTattE_provider_class_sequence.ending				
Underscore delimited parts of the filename	libTattE	provider	class	sequence	ending
Description	First part of the name, required to be this.	Single word name of the main provider EXAMPLE: Choice	Function classes supported in Table 2. EXAMPLE: D	A two digit decimal identifier to start at 00 and increment by 1 every time a library is sent to NIST. EXAMPLE: 07	.so
Example	libTattE_Choice_D_07.so				

1.18.2 Configuration and developer-defined data

The implementation under test may be supplied with configuration files and supporting data files. NIST will report the size of the supplied configuration files.

1.18.3 Submission folder hierarchy

Participant submissions should contain the following folders at the top level

- lib/ - contains all participant-supplied software libraries
- config/ - contains all configuration and developer-defined data
- doc/ - contains any participant-provided documentation regarding the submission
- validation/ - contains validation output

1.18.4 Installation and Usage

The implementation shall be installable using simple file copy methods. It shall not require the use of a separate installation program and shall be executable on any number of machines without requiring additional machine-specific license control procedures or activation. The implementation shall not use nor enforce any usage controls or limits based on licenses, number of executions, presence of temporary files, etc. It shall remain operable with no expiration date.

Hardware (e.g. USB) activation dongles are not acceptable.

1.18.5 Modes of operation

Implementations shall not require NIST to switch “modes” of operation or algorithm parameters. For example, the use of two different feature extractors must either operate automatically or be split across two separate library submissions.

1.19 Runtime behavior

1.19.1 Interactive behavior, stdout, logging

The implementation will be tested in non-interactive “batch” mode (i.e. without terminal support). Thus, the submitted library shall:

- Not use any interactive functions such as graphical user interface (GUI) calls, or any other calls, which require terminal interaction e.g. reads from “standard input”.
- Run quietly, i.e. it should not write messages to "standard error" and shall not write to “standard output”.
- Only if requested by NIST for debugging, include a logging facility in which debugging messages are written to a log file whose name includes the provider and library identifiers and the process PID. Please do not enable this by default.

1.19.2 Exception Handling

The application should include error/exception handling so that in the case of a fatal error, the return code is still provided to the calling application.

1.19.3 External communication

Processes running on NIST hosts shall not affect the runtime environment in any manner, except for memory allocation and release. Implementations shall not write any data to external resource (e.g. server, file, connection, or other process), nor read from such. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

1.19.4 Stateless behavior

All components in this test shall be stateless. Thus, all functions should give identical output, for a given input, independent of the runtime history. NIST will institute appropriate tests to detect stateful behavior. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

1.20 Single-thread Requirement

Implementations must run in single-threaded mode, because NIST will parallelize the test by dividing the workload across many cores and many machines simultaneously.

1.21 Time limits

The elemental functions of the implementations shall execute under the time constraints of Table 5. These time limits apply to the function call invocations defined in Table 5. Assuming the times are random variables, NIST cannot regulate the maximum value, so the time limits are 90-th percentiles. This means that 90% of all operations should take less than the identified duration.

The time limits apply per image. When K tattoo images of a subject are present, the time limits shall be increased by a factor K. To allow for diversity of algorithms, the time limit to conduct a search has been increased (see table below). NIST will explore the accuracy vs. speed tradeoff for tattoo recognition algorithms running on a fixed platform. Both accuracy and speed of the implementations tested will be

reported. A number of methods on performing efficient search are available in the literature[4], and sub-linear search implementations are encouraged.

Table 5 – Processing time limits (1 core) in seconds, per 640 x 480 image

	D	I
Function	Detection and Localization	1:N identification
Feature extraction for enrollment and identification	5	5
Identification of one search template against 100,000 single-image tattoo records.		~~16~~ 300
Enrollment finalization of 100,000 single-image tattoo records (including disk IO time)		720

1.22 Ground truth integrity

Some of the test data is derived from operational systems and may contain ground truth errors in which

- a single tattoo is present under two different identifiers, or
- two different tattoos are present under one identifier, or
- in which a tattoo is not present in the image.

If these errors are detected, they will be removed. NIST will use aberrant scores (high impostor scores, low genuine scores) to detect such errors. This process will be imperfect, and residual errors are likely. For comparative testing, identical datasets will be used and the presence of errors should give an additive increment to all error rates. For very accurate implementations this will dominate the error rate. NIST intends to attach appropriate caveats to the accuracy results. For prediction of operational performance, the presence of errors gives incorrect estimates of performance.

[4] Jégou, H., Douze, M., & Schmid, C. (2011). Product quantization for nearest neighbor search. In *IEEE PAMI*.

2. Data structures supporting the API

2.1 Data structures

2.1.1 Overview

In this test, a tattoo is represented by K ≥ 1 two-dimensional tattoo images.

2.1.2 Data structures for encapsulating multiple images

Some of the proposed datasets includes K > 2 same tattoo images per person for some persons. This affords the possibility to model a recognition scenario in which a new image of a tattoo is compared against all prior images. Use of multiple images per person has been shown to elevate accuracy over a single image for other biometric modalities.

For tattoo recognition in this test, NIST will enroll K ≥ 1 images for each unique tattoo. Both enrolled gallery and probe samples may consist of multiple images such that a template is the result of applying feature extraction to a set of K ≥ 1 images and then integrating information from them. An algorithm might fuse K feature sets into a single model or might simply maintain them separately. In any case the resulting proprietary template is contained in a contiguous block of data. All identification functions operate on such multi-image templates.

The number of images per unique tattoo will vary, and images may not be acquired uniformly over time. NIST currently estimates that the number of images K will never exceed 100. For the Tatt-E API, K of the same tattoo images of an individual are contained in data structure of Section 2.1.2.2.

2.1.2.1 TattE::Image Struct Reference

Struct representing a single image.

Public Member Functions

- **Image** ()
- **Image** (uint16_t widthin, uint16_t heightin, uint8_t depthin, ImageType typein, std::shared_ptr<uint8_t> datain)

Public Attributes

- uint16_t **width**
  Number of pixels horizontally.
- uint16_t **height**
  Number of pixels vertically.
- uint16_t **depth**
  Number of bits per pixel. Legal values are 8 and 24.
- ImageType ***imageType***
  Label describing the type of image.
- std::shared_ptr<uint8_t> **data**
  Managed pointer to raster scanned data. Either RGB color or intensity. If image_depth == 24 this points to 3WH bytes RGBRGBRGB... If image_depth == 8 this points to WH bytes IIIIIII.

2.1.2.2 TattE::MultiTattoo Typedef Reference

typedef std::vector< **Image** > **MultiTattoo**

Data structure representing a set of the same tattoo images from a single person.

2.1.3 Data Structure for detected tattoo

Implementations shall return bounding box coordinates of each detected tattoo in an image.

2.1.3.1 TattE::BoundingBox Struct Reference

Structure for bounding box around a detected tattoo.

Public Member Functions

- **BoundingBox** ()
- **BoundingBox** (uint16_t xin, uint16_t yin, uint16_t widthin, uint16_t heightin, double confin)

Public Attributes

- uint16_t **x**
  X-coordinate of top-left corner of bounding box around tattoo.
- uint16_t **y**
  Y-coordinate of top-left corner of bounding box around tattoo.
- uint16_t **width**
  Width, in pixels, of bounding box around tattoo.
- uint16_t **height**
  Height, in pixels, of bounding box around tattoo.
- double **confidence**
  Certainty that this region contains a tattoo. This value shall be on [0, 1]. The higher the value, the more certain.

2.1.4 Class for representing a tattoo in a MultiTattoo

2.1.4.1 TattE::TattooRep Class Reference

Class representing a tattoo or sketch template from image(s)

Public Member Functions

- **TattooRep** ()
  Default Constructor.
- void **addBoundingBox** (const **BoundingBox** &bb)
  This function should be used to add bounding box entries for each input image provided to the implementation for template generation. If there are 4 images in the MultiTattoo vector, then the size of boundingBoxes shall be 4. boundingBoxes[i] is associated with MultiTattoo[i].
- std::shared_ptr< uint8_t > **resizeTemplate** (uint64_t size)
  This function takes a size parameter and allocates memory of size and returns a managed pointer to the newly allocated memory for implementation manipulation. This class will take care of all memory allocation and de-allocation of its own memory. The implementation shall not de-allocate memory created by this class.
- const std::shared_ptr< uint8_t > **getTattooTemplatePtr** () const
- uint64_t **getTemplateSize** () const
  This function returns the size of the template data.
- std::vector< **BoundingBox** > **getBoundingBoxes** () const
  This function returns the bounding boxes for detected tattoos associated with the input images.

Private Attributes

- std::shared_ptr< uint8_t > **tattooTemplate**
  Proprietary template data representing a tattoo in images(s)
- uint64_t **templateSize**
  Size of template.
- std::vector< **BoundingBox** > **boundingBoxes**
  Data structure for capturing bounding boxes around the detected tattoo(s)

2.1.5 Data structure for result of an identification search

All identification searches shall return a candidate list of a NIST-specified length. The list shall be sorted with the most similar matching entries listed first with lowest rank.

2.1.5.1 TattE::Candidate Struct Reference

Data structure for result of an identification search.

Public Member Functions

- **Candidate** ()
- **Candidate** (bool assignedin, std::string idin, double scorein)

Public Attributes

- bool **isAssigned**
  If the candidate is valid, this should be set to true. If the candidate computation failed, this should be set to false.
- std::string **templateId**
  The template ID from the enrollment database manifest.
- double **similarityScore**
  Measure of similarity between the identification template and the enrolled candidate. Higher scores mean more likelihood that the samples are of the same person. An algorithm is free to assign any value to a candidate. The distribution of values will have an impact on the appearance of a plot of false-negative and false-positive identification rates.

2.1.6 Data Structure for return value of API function calls

2.1.6.1 TattE::ReturnStatus Struct Reference

A structure to contain information about the success/failure by the software under test. An object of this class allows the software to return some information from a function call. The string within this object can be optionally set to provide more information for debugging etc. The status code will be set by the function to Success on success, or one of the other codes on failure.

Public Member Functions

- **ReturnStatus** ()
- **ReturnStatus** (const TattE::ReturnCode **code**, const std::string **info=""**)
  *Create a **ReturnStatus** object.*

Public Attributes

- TattE::ReturnCode **code**
  Return status code.
- std::string **info**
  Optional information string.

2.1.7 Enumeration Type Documentation

2.1.7.1 enum TattE::ReturnCode[strong]

Return codes for the functions specified by this API.

Enumerator

Success Success

ConfigError Error reading configuration files

ImageTypeNotSupported Image type, e.g., sketches, is not supported by the implementation

RefuseInput Elective refusal to process the input

ExtractError Involuntary failure to process the image

ParseError Cannot parse the input data

TemplateCreationError Elective refusal to produce a template

EnrollDirError An operation on the enrollment directory failed (e.g. permission, space)

NumDataError The implementation cannot support the number of input images

TemplateFormatError One or more template files are in an incorrect format or defective

InputLocationError Cannot locate the input data - the input files or names seem incorrect

VendorError Vendor-defined failure

2.1.7.2 enum TattE::TemplateRole[strong]

Labels describing the type/role of the template to be generated (provided as input to template generation)

Enumerator

Enrollment Enrollment template used to enroll into gallery

Identification Identification template used for search

2.1.7.3 enum TattE::ImageType[strong]

Labels describing the image type.

Enumerator

Tattoo Tattoo image

Sketch Sketch of tattoo

Unknown Unknown or unspecified

2.2 File structures for enrolled template collection

An implementation converts a **MultiTattoo** into a template, using, for example the **createTemplate()** function of section 3.4.1.5.2. To support the Class I identification functions of Table 2, NIST will concatenate enrollment templates into a single large file, the EDB (for enrollment database). The EDB is a simple binary concatenation of proprietary templates. There is no header. There are no delimiters. The EDB may be hundreds of gigabytes in length.

This file will be accompanied by a manifest; this is an ASCII text file documenting the contents of the EDB. The manifest has the format shown as an example in Table 6. If the EDB contains N templates, the manifest will contain N lines. The fields are space (ASCII decimal 32) delimited. There are three fields. Strictly speaking, the third column is redundant.

Important: If a call to the template generation function fails, or does not return a template, NIST will include the Template ID in the manifest with size 0. Implementations must handle this appropriately.

Table 6 – Enrollment dataset template manifest

Field name	Template ID	Template Length	Position of first byte in EDB
Datatype required	std::string	Unsigned decimal integer	Unsigned decimal integer
Example lines of a manifest file appear to the right. Lines 1, 2, 3 and N appear.	90201744	1024	0
	Tattoo01	1536	1024
	7456433	512	2560
	...		
	Tattoo12	1024	307200000

The EDB scheme avoids the file system overhead associated with storing millions of individual files.

3. API Specification

The function prototypes from this document and any other supporting code will be provided in a "tatte.h" file made available to implementers via https://github.com/usnistgov/tattoo.

3.1 Namespace

All data structures and API interfaces/function calls will be declared in the `TattE` namespace.

3.2 Overview

This section describes separate APIs for the core tattoo applications described in section 1.10. All submissions to Tatt-E shall implement the functions required by the rules for participation listed before Table 2. Tatt-E participants shall implement the relevant C++ prototyped interfaces in this section. C++ was chosen in order to make use of some object-oriented features.

3.3 Detection and Localization (Class D)

This section defines an API for algorithms that can solely perform tattoo detection and localization. The detection task requires the implementation to detect whether an image contains a tattoo or not, and localization requires identifying the location of the tattoo within the image. Given an image, an implementation should

- For detection, classify whether a tattoo was detected in the image or not and provide a real-valued measure of detection confidence on [0,1], with 1 indicating absolute certainty that the image contains a tattoo and 0 indicating absolute certainty that the image does not contain a tattoo.
- For localization, report location(s) of one or more tattoos on different body locations in the form of a bounding box.

Table 7 – Procedural overview of the detection and localization test

Phase	Name	Description	Performance Metrics to be reported by NIST
Detection and Localization	Initialization	**initialize()** Give the implementation the name of a directory where any provider-supplied configuration data will have been placed by NIST. This location will otherwise be empty. The implementation is permitted read-only access to the configuration directory.	
	Detection	**detectTattoo()** For each of N images, pass single images to the implementation for tattoo detection. The implementation will set a boolean indicating whether a tattoo was detected or not and a detection certainty confidence score. Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.	Statistics of detection times. Accuracy metrics. The incidence of where the implementation failed to perform detection (non-successful return code).
	Localization	**localizeTattoos()** For each of N tattoo images, pass single images to the implementation for tattoo localization. The implementation will populate a vector with bounding boxes corresponding to the tattoos detected from the input image. Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.	Statistics of the time needed for this operation. Accuracy metrics. The incidence of where the implementation failed to perform localization.

3.3.1 TattE::DetectAndLocalizeInterface Class Reference

The interface to Class D implementations.

3.3.1.1 Public Member Functions

- virtual **~DetectAndLocalizeInterface** ()
- virtual **ReturnStatus initialize** (const std::string &configurationLocation)=0
  This function initializes the implementation under test. It will be called by the NIST application before any call to the functions detectTattoo and localizeTattoos().
- virtual **ReturnStatus detectTattoo** (const **Image** &inputImage, bool &tattooDetected, double &confidence)=0
  *This function takes an **Image** as input and indicates whether a tattoo was detected in the image or not.*
- virtual **ReturnStatus localizeTattoos**(const **Image** &inputImage, std::vector< **BoundingBox** > &boundingBoxes)=0
  *This function takes an **Image** as input, and populates a vector of **BoundingBox** with the number of tattoos detected on different body locations from the input image.*

3.3.1.2 Static Public Member Functions

- static std::shared_ptr< **DetectAndLocalizeInterface** > **getImplementation** ()
  *Factory method to return a managed pointer to the **DetectAndLocalizeInterface** object. This function is implemented by the submitted library and must return a managed pointer to the **DetectAndLocalizeInterface** object.*

3.3.1.3 Detailed Description

The interface to Class D implementations.

The class D detection and localization software under test must implement the interface **DetectAndLocalizeInterface** by subclassing this class and implementing each method specified therein.

3.3.1.4 Constructor & Destructor Documentation

- **virtual TattE::DetectAndLocalizeInterface::DetectAndLocalizeInterface ()[inline], [virtual]**

3.3.1.5 Member Function Documentation

3.3.1.5.1 virtual ReturnStatus TattE::DetectAndLocalizeInterface::initialize (const std::string & configurationLocation)[pure virtual]

This function initializes the implementation under test. It will be called by the NIST application before any call to the functions detectTattoo and localizeTattoos.

Parameters:

in	*configurationLocation*	A read-only directory containing any developer-supplied configuration parameters or run-time data. The name of this directory is assigned by NIST, not hardwired by the provider. The names of the files in this directory are hardwired in the implementation and are unrestricted.

3.3.1.5.2 virtual ReturnStatus TattE::DetectAndLocalizeInterface::detectTattoo (const Image & inputImage, bool & tattooDetected, double & confidence)[pure virtual]

This function takes an **Image** as input and indicates whether a tattoo was detected in the image or not.

Parameters:

in	*inputImage*	An instance of an **Image** struct representing a single image
out	*tattooDetected*	true if a tattoo is detected in the image; false otherwise

out	*confidence*	A real-valued measure of tattoo detection confidence on [0,1]. A value of 1 indicates certainty that the image contains a tattoo, and a value of 0 indicates certainty that the image does not contain a tattoo.

3.3.1.5.3 virtual ReturnStatus TattE::DetectAndLocalizeInterface::localizeTattoos(const Image & inputImage, std::vector< BoundingBox > & boundingBoxes, std::vector< BodyLocation > & bodyLocations)[pure virtual]

This function takes an **Image** as input, and populates a vector of **BoundingBox** with the number of tattoos detected on different body locations from the input image.

Parameters:

in	*inputImage*	An instance of an **Image** struct representing a single image
out	*boundingBoxes*	For each tattoo detected in the image, the function shall create a **BoundingBox**, populate it with a confidence score, the x, y, width, height of the bounding box, and add it to the vector.

3.3.1.6 static std::shared_ptr<DetectAndLocalizeInterface> TattE::DetectAndLocalizeInterface::getImplementation ()[static]

Factory method to return a managed pointer to the **DetectAndLocalizeInterface** object.

This function is implemented by the submitted library and must return a managed pointer to the **DetectAndLocalizeInterface** object.

Note:

A possible implementation might be: `return(std::make_shared<ImplementationD>());`

3.4 Identification (Class I)

The 1:N application proceeds in two phases, enrollment and identification. The identification phase includes separate pre-search feature extraction stage, and a search stage.

The design reflects the following *testing* objectives for 1:N implementations.

- support distributed enrollment on multiple machines, with multiple processes running in parallel
- allow recovery after a fatal exception, and measure the number of occurrences
- allow NIST to copy enrollment data onto many machines to support parallel testing
- respect the black-box nature of biometric templates
- extend complete freedom to the provider to use arbitrary algorithms
- support measurement of duration of core function calls
- support measurement of template size

Table 8 – Procedural overview of the identification test

Phase	#	Name	Description	Performance Metrics to be reported by NIST
Enrollment	E1	Initialization	**initializeEnrollmentSession()** Give the implementation the name of a directory where any provider-supplied configuration data will have been placed by NIST. This location will otherwise be empty.	

	E2	Parallel Enrollment	**createTemplate(TemplateRole=Enrollment)** The input will be one or more of the same tattoo image. This function will pass the input to the implementation for conversion to a single template. The implementation will return a template to the calling application. NIST's calling application will be responsible for storing all templates as binary files. These will not be available to the implementation during this enrollment phase. Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers. The same tattoo will not be enrolled twice.	Statistics of the times needed to enroll a tattoo. Statistics of the sizes of created templates. The incidence of failed template creations.
	E3	Finalization	**finalizeEnrollment()** Permanently finalize the enrollment directory. This supports, for example, adaptation of the image-processing functions, adaptation of the representation, writing of a manifest, indexing, and computation of statistical information over the enrollment dataset. The implementation is permitted **read-write-delete access** to the enrollment directory during this phase.	Size of the enrollment database as a function of population size N and the number of images. Duration of this operation. The time needed to execute this function shall be reported with the preceding enrollment times.
Pre-search	S1	Initialization	**initializeProbeTemplateSession()** Tell the implementation the location of an enrollment directory. The implementation could look at the enrollment data. Implementation initialize in preparation for search template creation. The implementation is permitted **read-only access** to the enrollment directory during this phase.	Statistics of the time needed for this operation. Statistics of the time needed for this operation.
	S2	Template preparation	**createTemplate(TemplateRole=Identification)** For each probe, create a template from a set of input tattoo(s) or a sketch image. This operation will generally be conducted in a separate process invocation to step S3. The implementation is **permitted no access** to the enrollment directory during this phase. The result of this step is a search template.	Statistics of the time needed for this operation. Statistics of the size of the search template.
Search	S3	Initialization	**initializeIdentificationSession()** Tell the implementation the location of an enrollment directory. The implementation should read all or some of the enrolled data into main memory, so that searches can commence. The implementation is permitted **read-only access** to the enrollment directory during this phase.	Statistics of the time needed for this operation.
	S4	Search	**identifyTemplate()** A template is searched against the enrollment database. The implementation is permitted **read-only access** to the enrollment directory during this phase.	Statistics of the time needed for this operation. Accuracy metrics - Type I + II error rates. Failure rates.

3.4.1 TattE::IdentificationInterface Class Reference

3.4.1.1 Public Member Functions

- virtual **~IdentificationInterface** ()
- virtual **ReturnStatus initializeEnrollmentSession** (const std::string &configurationLocation)=0
  This function initializes the implementation under test and sets all needed parameters.
- virtual **ReturnStatus createTemplate** (const **MultiTattoo** &inputTattoos, const **TemplateRole** &templateType, **TattooRep** &tattooTemplate, std::vector<**double**> &quality)=0

*This function takes a MultiTattoo and outputs a **TattooRep** object (essentially a template).*

- virtual **ReturnStatus finalizeEnrollment** (const std::string &enrollmentDirectory, const std::string &edbName, const std::string &edbManifestName)=0
  This function will be called after all enrollment templates have been created and freezes the enrollment data. After this call the enrollment dataset will be forever read-only.
- virtual **ReturnStatus initializeProbeTemplateSession** (const std::string &configurationLocation, const std::string &enrollmentDirectory)=0
  Before MultiTattoos are sent to the search template creation function, the test harness will call this initialization function.
- virtual **ReturnStatus initializeIdentificationSession** (const std::string &configurationLocation, const std::string &enrollmentDirectory)=0
  This function will be called once prior to one or more calls to identifyTemplate. The function might set static internal variables so that the enrollment database is available to the subsequent identification searches.
- virtual **ReturnStatus identifyTemplate** (const **TattooRep** &idTemplate, const uint32_t candidateListLength, std::vector< **Candidate** > &candidateList)=0
  This function searches an identification template against the enrollment set, and outputs a vector containing candidateListLength Candidates.

3.4.1.2 Static Public Member Functions

- static std::shared_ptr< **IdentificationInterface** > **getImplementation** ()
  *Factory method to return a managed pointer to the **IdentificationInterface** object.*

3.4.1.3 Detailed Description

The interface to Class I implementations.

The Class I submission software under test will implement this interface by subclassing this class and implementing each method therein.

3.4.1.4 Constructor & Destructor Documentation

- **virtual TattE::IdentificationInterface::~IdentificationInterface ()[inline], [virtual]**

3.4.1.5 Member Function Documentation

3.4.1.5.1 virtual ReturnStatus TattE::IdentificationInterface::initializeEnrollmentSession (const std::string & *configurationLocation*)[pure virtual]

This function initializes the implementation under test and sets all needed parameters.

This function will be called N=1 times by the NIST application, prior to parallelizing M >= 1 calls to createTemplate() via fork().

Parameters:

in	*configurationLocation*	A read-only directory containing any developer-supplied configuration parameters or run-time data files.

3.4.1.5.2 virtual ReturnStatus TattE::IdentificationInterface::createTemplate (const MultiTattoo & *inputTattoos*, const TemplateRole & *templateType*, TattooRep & *tattooTemplate, std::vector<double> & quality*)[pure virtual]

This function takes a MultiTattoo and outputs a **TattooRep** object (essentially a template) and a vector of quality values associated with each tattoo image.

For enrollment templates: If the function executes correctly (i.e. returns a successful exit status), the NIST calling application will store the template. The NIST application will concatenate the templates and pass the result to the enrollment finalization function. When the implementation fails to produce a template, it shall still

return a blank template (which can be zero bytes in length). The template will be included in the enrollment database/manifest like all other enrollment templates, but is not expected to contain any feature information.

For identification templates: If the function returns a non-successful return status, the output template will be not be used in subsequent search operations.

Parameters:

in	*inputTattoos*	An instance of a MultiTattoo structure. Implementations must alter their behavior according to the type and number of images/type of image contained in the structure. The input image type could be a tattoo or a sketch image. The MultiTattoo will always contain the same type of imagery, i.e., no mixing of tattoos and sketch images will occur. **Note that implementation support for sketch images is OPTIONAL. Implementation shall return TattE::ImageType::ImageTypeNotSupported if they do not support sketch images. All algorithms must support tattoo images.**
in	*templateType*	A value from the TemplateRole enumeration that indicates the intended usage of the template to be generated. In this case, either an enrollment template used for gallery enrollment or an identification template used for search.
out	*tattooTemplate*	Tattoo template object. For each tattoo detected in the MultiTattoo, the function shall provide the bounding box coordinates in each image. The bounding boxes shall be captured in the TattooRep.boundingBoxes variable, which is a vector of **BoundingBox** objects. If there are 4 images in the MultiTattoo vector, then the size of boundingBoxes shall be 4. boundingBoxes[i] is associated with MultiTattoo[i].
out	*quality*	A vector of quality values, one for each input tattoo image. This will be an empty vector when passed into this function, and the implementation shall populate a quality value corresponding to each input image. quality[i] shall correspond to inputTattoos[i]. A measure of tattoo quality on [0,1] is indicative of expected utility to the matcher, or matchability. This value could measure tattoo distinctiveness/information richness, and would be an indicator of how well the tattoo would be expected to match. A value of 1 indicates high quality and that the tattoo would be expected to match well, and a value of 0 means low quality, indicative that tattoo would not match well.

3.4.1.5.3 virtual ReturnStatus TattE::IdentificationInterface::finalizeEnrollment (const std::string & *enrollmentDirectory*, const std::string & *edbName*, const std::string & *edbManifestName*)[pure virtual]

This function will be called after all enrollment templates have been created and freezes the enrollment data. After this call the enrollment dataset will be forever read-only.

This function allows the implementation to conduct, for example, statistical processing of the feature data, indexing and data re-organization. The function may create its own data structure. It may increase or decrease the size of the stored data. No output is expected from this function, except a return code. The function will generally be called in a separate process after all the enrollment processes are complete. NOTE: Implementations shall not move the input data. Implementations shall not point to the input data. Implementations should not assume the input data would be readable after the call. Implementations must, **at a minimum, copy the input data** or otherwise extract what is needed for search.

Parameters:

in	*enrollmentDirectory*	The top-level directory in which enrollment data was placed. This variable allows an implementation to locate any private initialization data it elected to place in the directory.
in	*edbName*	The name of a single file containing concatenated templates, i.e. the EDB described in *Data Structures Supporting the API*. While the file will have read-write-delete permission, the implementation should only alter the file if it preserves the necessary content, in other files for example. The file may be opened directly. It is not necessary to prepend a directory name. This is a NIST-provided input - implementers shall not internally hard-code or assume any values.
in	*edbManifestName*	The name of a single file containing the EDB manifest described in *Data Structures Supporting the API* . The file may be opened directly. It is not necessary to prepend a directory name. This is a NIST-provided input - implementers shall not internally hard-code or assume any values.

3.4.1.5.4 virtual ReturnStatus TattE::IdentificationInterface::initializeProbeTemplateSession (const std::string & *configurationLocation*, const std::string & *enrollmentDirectory*)[pure virtual]

Before MultiTattoos are sent to the search template creation function, the test harness will call this initialization function.

This function initializes the implementation under test and sets all needed parameters. This function will be called N=1 times by the NIST application, prior to parallelizing M >= 1 calls to createTemplate() via fork(). Caution: The implementation should tolerate execution of P > 1 processes on the one or more machines each of which may be reading from this same enrollment directory in parallel. The implementation has read-only access to its prior enrollment data.

Parameters:

in	*configurationLocation*	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
in	*enrollmentDirectory*	The read-only top-level directory in which enrollment data was placed and then finalized by the implementation. The implementation can parameterize subsequent template production on the basis of the enrolled dataset.

3.4.1.5.5 virtual ReturnStatus TattE::IdentificationInterface::initializeIdentificationSession (const std::string & *configurationLocation*, const std::string & *enrollmentDirectory*)[pure virtual]

This function will be called once prior to one or more calls to identifyTemplate. The function might set static internal variables so that the enrollment database is available to the subsequent identification searches.

Parameters:

in	*configurationLocation*	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
in	*enrollmentDirectory*	The read-only top-level directory in which enrollment data was placed.

3.4.1.5.6 virtual ReturnStatus TattE::IdentificationInterface::identifyTemplate (const TattooRep & *idTemplate*, const uint32_t *candidateListLength*, std::vector< Candidate > & *candidateList*)[pure virtual]

This function searches an identification template against the enrollment set, and outputs a vector containing candidateListLength Candidates.

Each candidate shall be populated by the implementation and added to candidateList. Note that candidateList will be an empty vector when passed into this function. The candidates shall appear in descending order of similarity score - i.e. most similar entries appear first.

Parameters:

in	*idTemplate*	A template from **createTemplate()**. If the value returned by that function was non-successful, the contents of idTemplate will not be used, and this function will not be called.
in	*candidateListLength*	The number of candidates the search should return.
out	*candidateList*	Each candidate shall be populated by the implementation. The candidates shall appear in descending order of similarity score - i.e. most similar entries appear first.

3.4.1.5.7 static std::shared_ptr<IdentificationInterface> TattE::IdentificationInterface::getImplementation ()[static]

Factory method to return a managed pointer to the **IdentificationInterface** object.

This function is implemented by the submitted library and must return a managed pointer to the **IdentificationInterface** object.

Note:

A possible implementation might be: `return (std::make_shared<ImplementationI>());`

Annex A
Submissions of Implementations to Tatt-E

A.1 Submission of implementations to NIST

NIST requires that all software, data and configuration files submitted by the participants be signed and encrypted. Signing is done with the participant's private key, and encryption is done with the NIST public key. The detailed commands for signing and encrypting are given here: https://www.nist.gov/itl/iad/image-group/products-and-services/encrypting-softwaredata-transmission-nist.

NIST will validate all submitted materials using the participant's public key, and the authenticity of that key will be verified using the key fingerprint. This fingerprint must be submitted to NIST by writing it on the signed Tatt-E Participation Agreement that is published on the Tatt-E website - https://www.nist.gov/programs-projects/tattoo-recognition-technology-evaluation-tatt-e.

By encrypting the submissions, we ensure privacy; by signing the submission, we ensure authenticity (the software actually belongs to the submitter). NIST will reject any submission that is not signed and encrypted. NIST accepts no responsibility for anything that is transmitted to NIST that is not signed and encrypted with the NIST public key.

A.2 How to participate

Those wishing to participate in Tatt-E testing must do all of the following, on the schedule listed on Page 2.

- IMPORTANT: Follow the instructions for cryptographic protection of your software and data here - https://www.nist.gov/itl/iad/image-group/products-and-services/encrypting-softwaredata-transmission-nist
- Send a signed and fully completed copy of the *Application to Participate in the Tattoo Recognition Technology - Evaluation (Tatt-E)* ~~contained in this document~~ that is published on the Tatt-E website. This must identify, and include signatures from, the Responsible Parties as defined in the application. The properly signed Tatt-E Application to Participate shall be sent to NIST as a PDF.
- Provide a software library that complies with the API (Application Programmer Interface) specified in this document.
    - Encrypted data and libraries below 20MB can be emailed to NIST at tatt-e@nist.gov.
    - Encrypted data and libraries above 20MB shall be

      EITHER
        - Split into sections AFTER the encryption step. Use the unix "split" commands to make 9MB chunks, and then rename to include the filename extension need for passage through the NIST firewall.
        - you% split –a 3 –d –b 9000000 libTattE_Choice_D_07.tgz.gpg
        - you% ls -1 x??? | xargs –iQ mv Q libTattE_Choice_D_07_Q.tgz.gpg
        - Email each part in a separate email. Upon receipt NIST will
        - nist% cat tatte_choice_D07_*.tgz.gpg > libTattE_Choice_D_07.tgz.gpg

      OR
        - Made available as a file.zip.gpg or file.zip.asc download from a generic http webserver[5],

      OR
        - Mailed as a file.zip.gpg or file.zip.asc on CD / DVD to NIST at this address:

Tatt-E Test Liaison (A210) 100 Bureau Drive A210/Tech225/Stop 8940 NIST	In cases where a courier needs a phone number, please use NIST shipping and handling on: 301 -- 975 -- 6296.

[5] NIST will not register, or establish any kind of membership, on the provided website.

Gaithersburg, MD 20899-8940 USA	

A.3 Implementation validation

Registered Participants will be provided with a small validation dataset and test program upon NIST receipt of either a signed Participation Agreement or an email to tatt-e@nist.gov indicating participation intentions. Instructions on obtaining the validation package will be emailed to participants thereafter.

~~via https://github.com/usnistgov/tattoo shortly after the final evaluation plan is released. An announcement will be made on the Tatt-E website when the validation package is available.~~

The validation test programs shall be compiled by the provider. The output of these programs shall be submitted to NIST.

Prior to submission of the software library and validation data, the Participant must verify that their software executes on the validation images and produces correct scores and templates.

Software submitted shall implement the Tatt-E API Specification as detailed in the body of this document.

Upon receipt of the software library and validation output, NIST will attempt to reproduce the same output by executing the software on the validation imagery, using a NIST computer. In the event of disagreement in the output, or other difficulties, the Participant will be notified.

Please note that the provided validation software and imagery is meant only for validation purposes and does not reflect how NIST will perform actual testing. The validation images are not representative of the actual test data that will be used to evaluate implementations.